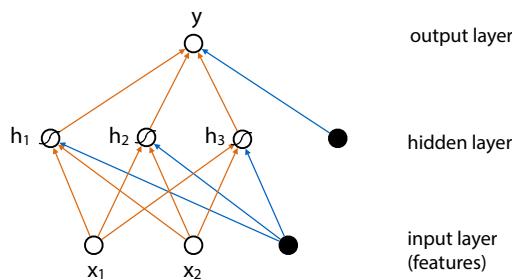


Deep Learning

Machine Learning 2020
mlvu.github.io

recap



2

We are picking up the discussion about neural networks from the last lecture. A neural network is model described by a graph: each node represents a scalar value. The value of each node is computed from the incoming edges by multiplying the weight on the edge by the value of the node it connects to.

We train the model by tuning the weights.

The feedforward network is the simplest way of wiring up a neural net, but we will see other possibilities later.

Every orange and blue line in this picture represents one parameter of the model.

Since this lecture is mostly about extending the backpropagation algorithm from the last lecture, we will review it briefly.

1974, 1980's: backpropagation

Break your computation down into a chain of *modules*.

Work out the derivative of each module with respect to its input **symbolically**.

Compute the global gradient for a given input by multiplying these gradients.

3

example

$$f(x) = \frac{2}{\sin(e^{-x})} \quad f(x) = d(c(b(a(x))))$$

modules:

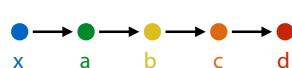
$$d(c) = \frac{2}{c}$$

$$c(b) = \sin b$$

$$b(a) = e^a$$

$$a(x) = -x$$

computation graph:



To show that backpropagation is a generic algorithm for working out gradients, we'll first show how it works for some arbitrary function. Defining the sub functions a , b , c , and d as shown, we can write $f(x) = d(c(b(a(x))))$.

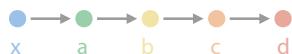
The graph on the right is a **computation graph**: each node represents a small computer program that receives an input, computes an output and passes it on to another module.

Normally, we wouldn't break a function up in such small modules. This is just a simple example to illustrate the principle.

4

example

$$f(x) = \frac{2}{\sin(e^{-x})}$$



$$d(c) = \frac{2}{c}$$

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x}$$

global derivative

local derivatives

$$c(b) = \sin b$$

$$b(a) = e^a$$

$$a(x) = -x$$

We'll call the derivative of the whole function the **global derivative**, and the derivative of each module with respect to its input we will call a **local derivative**.

5

backpropagation

Write your function as a composition of **modules**.

What the modules are up to you.

Work out the local derivative of each module **symbolically**.

Do a **forward pass** for a given input **x**.

i.e. compute the function $f(x)$, remember the intermediate values

Compute the local derivatives for **x**, and multiply to compute the global derivative.

More fine-grained modules make the local derivatives easier to work out, but increase numeric instability.

6

work out *local* derivatives

symbolically

$$d(c) = \frac{2}{c} \quad \frac{\partial f}{\partial x} = \frac{\partial d}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x}$$

$$c(b) = \sin b$$

$$b(a) = e^a \quad \frac{\partial f}{\partial x} = -\frac{2}{c^2} \cdot \cos b \cdot e^a \cdot -1$$

$$a(x) = -x$$

7

compute a forward pass ($x=-4.499$)

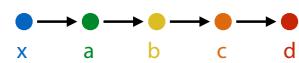
retain values of **a, b, c, d**

$$f(-4.499) = 2$$

$$d = \frac{2}{c} = 2$$

computation graph:

$$c = \sin b = 1$$



$$b = e^a = 90$$

$$a = -x = 4.499$$

8

compute the backward pass

numerically

$$f(-4.499) = 2 \quad \frac{\partial f}{\partial x} = -\frac{2}{c^2} \cdot \cos b \cdot e^a \cdot -1$$

$$d = \frac{2}{c} = 2 \quad = -\frac{2}{2^2} \cdot \cos 90 \cdot e^{4.499} \cdot -1$$

$$c = \sin b = 1 \quad = -\frac{1}{2} \cdot 0 \cdot 90 \cdot -1 = 0$$

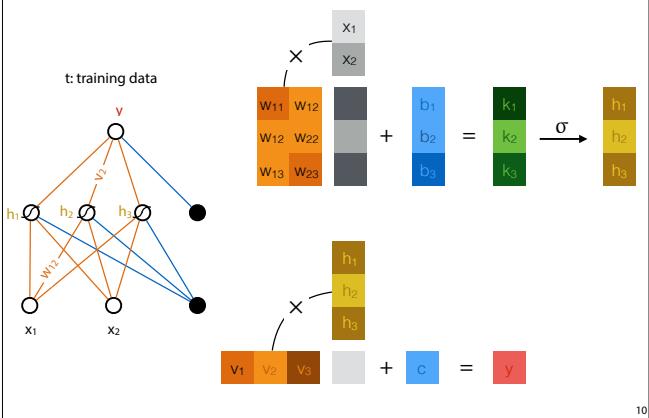
$$b = e^a = 90$$

$$a = -x = 4.499$$

Now we just fill in the intermediate values for the local derivatives, and compute the product numerically.

9

neural networks: it's all just linear algebra



10

Last lecture, we discussed neural networks from the perspective of a graphical model. Dots joined by arrows, to describe a computation.

However, there's another perspective that can greatly simplify things. Most of the operations used in neural networks can also be written as matrix multiplications.

$$f(x) = V \sigma(Wx + b) + c$$

As you can see, this greatly simplifies our notation. It also allows for very efficient implementation of neural networks, since matrix multiplication can be implemented very efficiently (especially on a GPU).

This is what we will discuss today: how to simplify the basic idea of neural networks into a very powerful and flexible framework for creating highly complex machine learning models.

deep learning

part 1:

Deep learning systems (aka autodiff engines)
tensors, matrix calculus, backpropagation revisited

Backpropagation revisited
Multivariate chain rule, calculus over tensors

part 2:

Making it work

vanishing gradients, minibatches, optimizers, regularization

Convolutional neural networks
aka ConvNets or CNNs

What is deep learning?

Autoencoders have been moved to "Deep learning 2"

12

aims

Scalar backpropagation -> Tensor backpropagation

Tensor operations (i.e. matrix multiplication) are easy to parallelise and run well on GPUs

Functions with multiple inputs and outputs

Backpropagation over any computation

No matter how complex

Ingredients: **tensors** and **functions over tensors**

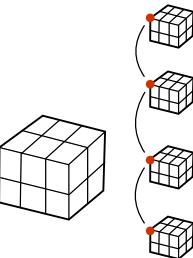
13

In order to scale up the basic principle of backpropagation on neural networks, we want to move from operations on scalars to the linear algebra view: everything is a tensor and all operations (including this in the backpropagation step) are tensor operations.

We will first define two ingredients: tensors, and functions over tensors.

tensors

scalar vector matrix



0-tensor

1-tensor

matrix

shape: 3

shape: 3x2

shape: 2x3x2

14

The main ingredient in the language of deep learning is the **tensor**. For our purposes, a tensor is nothing more than a straightforward generalisation of vectors and matrices to higher dimensionalities. We can deal with any kind of data so long as it can be cast as a tensor.

If you've done the first worksheet, you'll know the tensor as the basic data structure of numpy.

Let's look at some examples of how data is represented in tensor form.

data as tensor



181	46	male
181	50	male
166	44	female
171	38	female
152	36	female
156	40	female
167	40	female
170	45	male
178	50	male
191	50	male

X =

181	46
181	50
166	44
171	38
152	36
156	40
167	40
170	45
178	50
191	50
166	38

y =

0
0
1
1
1
1
1
0
0
0
1

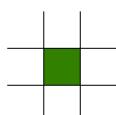
15

A simple dataset, with numeric features can simply be represented as a matrix (with a corresponding vector for the labels).

Any categoric features or labels should be converted to numeric features (normally by one-hot coding, see the Methodology 2 lecture)

images as 3-tensors

pixel



width height

channels



16

Images can be represented as 3-tensors. In an RGB image, the color of a single pixel is represented using three values between 0 and 1 (how red it is, how green it is and how blue it is). This means that an RGB image can be thought of as a stack of three color channels, represented by matrices.

This stack is a 3-tensor.

source: <http://www.adsell.com/scanning101.html>

image dataset: 4-tensor

```
In [5]: from keras.datasets import cifar10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train.shape
Out[5]: (50000, 32, 32, 3)
```

17

If we have a dataset of multiple images, we get a 4-tensor. The image dimensions we saw already, and one extra, indexing the different images in the data.

The Keras code shows how this looks in python: we load the CIFAR 10 dataset. The training data contains 50 000 images, each with 32 by 32 resolution and 3 color channels.

functions

Functions can have multiple **inputs** and **outputs**. All inputs, outputs are **tensors**.
Also called operations, or ops

Functions implement:

- **forward(...)**
computing the **outputs** given the **inputs**
- **backward(...)**
computing the **gradients over the inputs** given the **gradients over the outputs**

18

This is the basic principle of a deep learning system. We define modules that we chain together into a **computation graph**. The modules implement a local derivative (more on this later) in addition to the forward computation. This allows us to compute the gradient through backpropagation.

computation graph

$(a \times b) + b = c$

19

This is what we mean by a **computation graph**, a directed acyclic graph that shows the data (tensors **a**, **b**, **c**, **x**) flowing through the functions.

A deep learning system uses a computation graph to execute a given computation (**the forward pass**) and then uses the backpropagation algorithm to compute the gradients to the data nodes with respect to the output (**the backward pass**).

If we are not interested in the specifics of the function being applied, we will omit the circle (as we did in the previous lecture).

autodiff engine

Perform computation by chaining **functions**.

Keep track of all computation in a **computation graph**.

When the computation is finished, walk *backward* through the **computation graph** to perform **backpropagation**.

Eager evaluation: build the graph as we perform the computation.

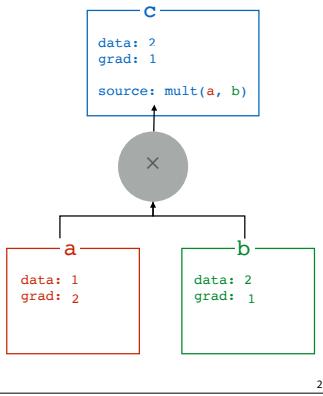
20

example

```
a = Tensor(data=1)
b = Tensor(data=2)

c = a * b

backprop(c)
```



In eager mode deep learning systems, we create a node in our computation graph (called a variable here) by specifying what data it should contain. The result is a tensor object that stores *both* the data, and the gradient over that data (which will be filled later).

Here we create the variables **a** and **b**. If we now apply an operation (a module) to these, for instance to multiply their values, the result has another variable **c**. We compute the data stored in **c** by running the computation, but we also store references to the variables that were used to create **c**, and the module that created it.

Using this graph, we can perform the back propagation from a given starting node. We work our way down the graph computing the derivative of each variable with respect to **c**.

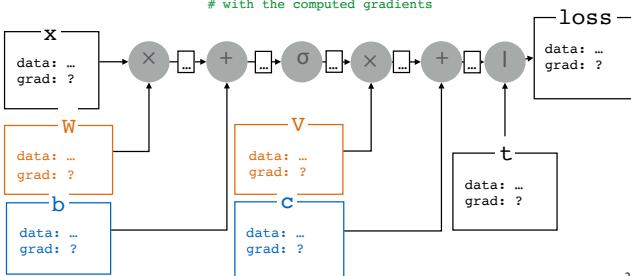
These names and this syntax are loosely inspired by those of PyTorch, but the concepts are similar for all deep learning frameworks.

```
loss( $\mathbf{V} \sigma(\mathbf{Wx} + \mathbf{b}) + \mathbf{c}, \mathbf{t}$ )
```

```
w, v, b, c = Tensor(...), Tensor(...), Tensor(...), Tensor(...)

for x, t in data:
    h = sigmoid(mult(w, x) + b)
    y = mult(v, h) + c
```

```
loss = l(y, t) # compute the loss
backprop(loss)
gradient_descent_step() # perform one step of GD
# with the computed gradients
```



Here's what a training loop would look like for a simple two-layer feedforward network. The computation graph shown below is rebuilt from scratch for every iteration of the training loop, and cleared at the end. The variables **W**, **V**, **b** and **c**, that define our neural network are re-used and updated every iteration

Note that the output of every module is also a variable, with its own data and its own gradient.

Note that the multiplications are now matrix multiplications

backpropagation

Functions can have any number of inputs and outputs.

Inputs and outputs can be tensors of any rank.

The final output must be a scalar.

That is, we always take the derivative of a scalar function.

Once we have the computation graph, we can use it to work out the gradient for all values in the graph. We will assume these three basic rules. The last one is essential: the function for which we compute the derivative (with respect to all values in the comp graph) **must** be a scalar. As we shall see, this will allow us to work backwards down the graph, computing the gradients.

Note that this does not prevent us from learning networks with multiple high-dimensional outputs. We just need to define a **single scalar loss** on that output.

backpropagation revisited

The multivariate chain rule

Dealing with multiple paths in the comp. graph

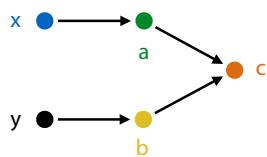
Backpropagation with tensors

Matrix calculus

In order to make backpropagation flexible and robust enough to work in this setting, we need to discuss two features that we haven't mentioned yet: how to perform backpropagation if the result depends on the dependent variable along different computation paths, and how to take derivatives when the variables aren't scalars.

24

multiple inputs

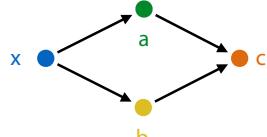


$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial a} \frac{\partial a}{\partial x} \quad \frac{\partial c}{\partial y} = \frac{\partial c}{\partial b} \frac{\partial b}{\partial y}$$

25

If a function has multiple inputs, there isn't usually a problem applying the chain rule. If we want the derivative with respect to x , we apply the chain rule over the path from x to c . If we want the derivative with respect to y , we apply the chain rule along the other path.

diamonds



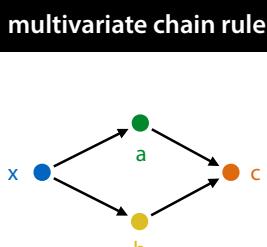
$$\frac{\partial c}{\partial x} = ? \quad \frac{\partial c}{\partial a} \frac{\partial a}{\partial x}$$

$$\frac{\partial c}{\partial x} = ? \quad \frac{\partial c}{\partial b} \frac{\partial b}{\partial x}$$

26

We only run into a problem if c has two inputs, both depending on w , we can't apply the chain rule. The chain rule we know only works for functions of one input (or one input we're interested in taking the derivative for).

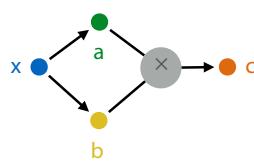
Do we apply the chain rule over a or over b ?



$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial a} \frac{\partial a}{\partial x} + \frac{\partial c}{\partial b} \frac{\partial b}{\partial x}$$

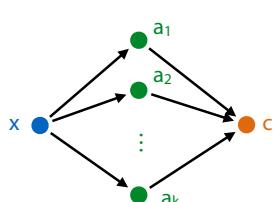
27

For such cases, we need the **multivariate chain rule**. It's very simple: to work out the derivative of a function with multiple inputs we just take a single derivative for each input (treating the others as constants), and sum them.



$$\begin{aligned}\frac{\partial c}{\partial x} &= \frac{\partial ab}{\partial x} = \frac{\partial ab}{\partial a} \frac{\partial a}{\partial x} + \frac{\partial ab}{\partial b} \frac{\partial b}{\partial x} \\ &= b \frac{\partial a}{\partial x} + a \frac{\partial b}{\partial x}\end{aligned}$$

28



$$\frac{\partial c}{\partial x} = \sum_i \frac{\partial c}{\partial a_i} \frac{\partial a_i}{\partial x}$$

29

backpropagation revisited

~~The multivariate chain rule~~
Dealing with multiple paths in the comp. graph

Backpropagation with tensors
Matrix calculus

In order to make backpropagation flexible and robust enough to work in this setting, we need to discuss two features that we haven't mentioned yet: how to perform backpropagation if the result depends on the dependent variable along different computation paths, and how to take derivatives when the variables aren't scalars.

30

matrix calculus

Can we express the local derivatives in terms of the tensor inputs and outputs?

$$\begin{aligned}l &= \sum (y - t)^2 \\ y &= Wh + c \\ h &= \sigma(k) \\ k &= Wx + b\end{aligned}\quad \frac{\partial l}{\partial W} = \frac{\partial l}{\partial y} \frac{\partial y}{\partial h} \frac{\partial h}{\partial k} \frac{\partial k}{\partial W}$$

31

We've worked the forward pass of a feedforward network as a series of matrix operations, but for backpropagation, everything is still defined in terms of scalars. We can work out the derivative for one element of W , but how do we express the gradient over the whole of W , in a single matrix operation?

In short, we would like to have a chain rule like the one shown on the right, and then to work out how to compute those local derivatives efficiently.

What does it mean to take the derivative of a vector, or with respect to a matrix?

scalar over vector

$$f(\mathbf{a}) = \mathbf{a}^T \mathbf{b}$$

$$\frac{\partial \mathbf{a}^T \mathbf{b}}{\partial \mathbf{a}}$$

$$\frac{\partial \mathbf{a}^T \mathbf{b}}{\partial a_3} = \frac{\partial a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4}{\partial a_3} = b_3$$

$$\frac{\partial \mathbf{a}^T \mathbf{b}}{\partial \mathbf{a}} = [b_1, b_2, b_3, b_4]^T$$

(this is just the gradient)

32

Let's start simple: say we have a function that takes a vector argument, and returns a scalar.

The main trick to working out what a matrix derivative means is always to think the derivative of one element in the output over one element in the input (just pick two random ones).

Once you've worked out what the scalar derivatives are, you can usually see how they form a matrix or vector together.

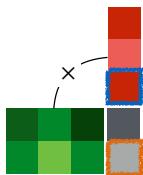
Normally, you need to worry about how exactly to shape such an output (like whether to form a column or a row vector), but in our case, as we'll see, we don't usually need to worry about that.

We're using a dot product just to show the principle on a very simple function, but this applies to any function that takes a vector and somehow turns it into a scalar.

vector over vector

$$f(\mathbf{a}) = \mathbf{B}\mathbf{a}$$

$$\frac{\partial \mathbf{B}\mathbf{a}}{\partial \mathbf{a}}$$



$$\frac{\partial (\mathbf{B}\mathbf{a})_2}{\partial a_3} = \frac{\partial \mathbf{B}_2 \cdot \mathbf{a}}{\partial a_3} = \frac{\partial \mathbf{B}_{21} a_1 + \mathbf{B}_{22} a_2 + \mathbf{B}_{23} a_3}{\partial a_3} = \mathbf{B}_{23}$$

$$\frac{\partial \mathbf{B}\mathbf{a}}{\partial \mathbf{a}} = \mathbf{B}$$

33

Here is another example: a function that takes a vector and produces a vector. We simplify by working out the derivative of the second element of the output over the third element of the input.

matrix calculus

Start with the scalar derivatives:
one output over one input

Tensor derivative:
arrange all possible scalar derivatives into a tensor

How to arrange?

34

	function returns a		
	scalar	vector	matrix
scalar	scalar	vector	matrix
vector	vector	matrix	?
matrix	matrix	?	?

$$\frac{\partial l}{\partial W} = \frac{\partial l}{\partial y} \frac{\partial y}{\partial h} \frac{\partial h}{\partial k} \frac{\partial k}{\partial W}$$

35

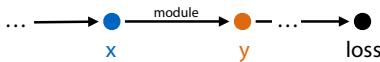
The matrix derivative is just an arrangement of every possible scalar derivative: every possible scalar element in the output over every possible scalar element in the input. This tells us what to expect: if we have a vector input and a vector output, all pairs of elements from the two are naturally arranged in a **matrix**. For certain combinations (vector/matrix, matrix/matrix) the results are tensors of order 3 or higher. You can still work these out, but the added complexity doesn't really pay off.

Luckily, we can use a trick to simplify things for ourselves: the *global* derivative that we are ultimately interested in is always in the leftmost column: because our computation always results in a scalar loss, we are ultimately only interested in the derivative of a scalar over an n-tensor of weights. The resulting derivative of that has the exact same shape as the weight tensor itself.

The intermediate (local) derivatives may result in complex

high-order tensors, but we can avoid those by not computing the local derivatives explicitly, but instead *accumulating them left to right*.

solution: accumulate the gradient product



`forward(x)`: given input x , compute output y .

$$\text{backward}(l_y): \text{given } l_y = \frac{\partial \text{loss}}{\partial y}$$

$$\text{compute } \frac{\partial \text{loss}}{\partial y} \frac{\partial y}{\partial x}$$

convention: the gradient of A has the same shape as A

36

So instead of letting the backward function compute the local derivative explicitly, we give it the derivative of the loss over the output, and ask it to multiply that with the local derivative of the output over the input.

This multiplication can be done without computing the local derivative explicitly. Let's look at an example.

for example

$$k = Wx + b$$

$$\frac{\partial l}{\partial W} = \frac{\partial l}{\partial y} \frac{\partial y}{\partial h} \frac{\partial h}{\partial k} \frac{\partial k}{\partial W}$$

`forward(W, x, b)`: compute $Wx + b$

`backward(l_k)`: compute

$$\frac{\partial l}{\partial k} \frac{\partial k}{\partial W}, \quad \frac{\partial l}{\partial k} \frac{\partial k}{\partial x}, \quad \frac{\partial l}{\partial k} \frac{\partial k}{\partial b}$$

This is the first layer of our feedforward network. It has three inputs W , x , and b , and one output k . That means the backward function is given the derivative of the loss over k , and should output the product of that derivative with three local derivative: k over W , k over x and k over b .

We'll look at the first as an example. The local derivative k over W is a 3 tensor which is difficult to work with, and would take up a large amount of memory, so we don't want to compute it explicitly. Let's see if we can work out a more efficient expression for the required product.

37

step one: work out the scalar derivative

$$\frac{\partial l}{\partial k} \frac{\partial k}{\partial W}$$

$$\frac{\partial l}{\partial k} \frac{\partial k}{\partial W_{23}}$$

As before, we'll first simplify, by looking at the scalar derivative for just one element of W . This doesn't get rid of the vector k , however: both sides of the product still represent multiple derivatives, and we haven't defined how to multiply them.

In this case they're vectors, and we know how to multiply vectors, but we'd like to derive a general approach that also works if k is complicated tensor that we don't know how to multiply. We can do this with the multivariate chain rule.

38

multivariate chain rule

$$\begin{aligned}
 \mathbf{k} &= \mathbf{W}\mathbf{x} + \mathbf{b} \\
 \frac{\partial l}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial W_{23}} &= \sum_i \frac{\partial l}{\partial k_i} \frac{\partial k_i}{\partial W_{23}} \\
 &= \sum_i \frac{\partial l}{\partial k_i} \frac{\partial (\mathbf{W}\mathbf{x} + \mathbf{b})_i}{\partial W_{23}} \\
 &= \sum_i \frac{\partial l}{\partial k_i} \frac{\partial (\mathbf{W}_{i\cdot} \mathbf{x})}{\partial W_{23}} \\
 &= \sum_i \frac{\partial l}{\partial k_i} \frac{\partial \sum_j (\mathbf{W}_{ij} x_j)}{\partial W_{23}} \\
 &= \sum_{i,j} \frac{\partial l}{\partial k_i} \frac{\partial (\mathbf{W}_{ij} x_j)}{\partial W_{23}} = \frac{\partial l}{\partial k_2} x_3
 \end{aligned}$$

If we think of this as a computation graph with scalar nodes, \mathbf{k} just represents different inputs to the function that ultimately computes l . That means that the derivative of l over W_{23} is just the sum of the derivatives through each element of \mathbf{k} . This works whatever the rank and shape of \mathbf{k} ; it could be a huge 9-tensor, and all we have to do is flatten it, and sum over its derivatives.

This is not an efficient way of actually computing the required product, it allows us to work out that the derivative of the loss with respect to element (2, 3) of \mathbf{W} is the product of element 2 of the incoming vector and element 3 of the input.

$$\frac{\partial l}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial W_{ij}} = \frac{\partial l}{\partial k_i} x_j$$

$$\begin{array}{c}
 \times^T \quad x_1 \quad x_2 \\
 \boxed{l} \quad \boxed{\frac{\partial l}{\partial \mathbf{k}}} \quad \boxed{\frac{\partial l}{\partial \mathbf{W}}}
 \end{array}$$

$$\frac{\partial l}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{W}} = \frac{\partial l}{\partial \mathbf{k}} \mathbf{x}^T$$

40

This means that we can compute the derivatives of all elements of the matrix \mathbf{W} (which is itself a matrix of the same size and shape as \mathbf{W}) by taking the **outer product** of the given vector \mathbf{l} over \mathbf{k} , and the input \mathbf{x} .

for example

$$\mathbf{k} = \mathbf{W}\mathbf{x} + \mathbf{b} \quad \frac{\partial l}{\partial \mathbf{W}} = \frac{\partial l}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{W}}$$

`forward(W, x, b)`: compute $\mathbf{W}\mathbf{x} + \mathbf{b}$

`backward(l_k)`: compute

$$\frac{\partial l}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{W}}, \quad \frac{\partial l}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{x}}, \quad \frac{\partial l}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{b}}$$

$$\frac{\partial l}{\partial \mathbf{k}} \mathbf{x}^T, \quad \mathbf{W}^T \frac{\partial l}{\partial \mathbf{k}}, \quad \frac{\partial l}{\partial \mathbf{k}}$$

41

The derivatives over \mathbf{x} and \mathbf{b} can be worked out in the same way.

in summary

model: the application of functions to tensors

each function defines a `backward()` function:
given the gradient over the **outputs**, compute the gradient over the **inputs**.

backpropagation walks backwards through the graph, accumulating the gradient product.

42

working out the backward function

Usually not necessary, only if you write your own module
But it's good to understand the principle

- Phrase the problem in terms of scalar derivatives: work out the derivative for one element of the input.
- Use the multivariate chain rule: sum over all elements of the output variable.
- Work out the general solution in terms of matrix operations

Further reading:

<https://compsci682-fa18.github.io/docs/vecDerivs.pdf>
<http://cs231n.stanford.edu/handouts/derivatives.pdf>

43

backpropagation revisited

~~The multivariate chain rule~~

Dealing with multiple paths in the comp. graph

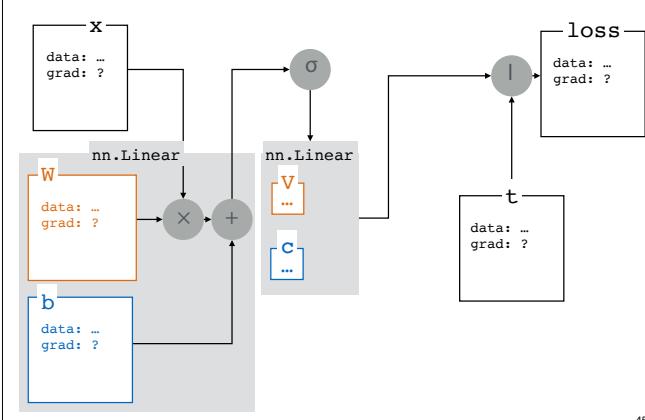
~~Backpropagation with tensors~~

Matrix calculus

In order to make backpropagation flexible and robust enough to work in this setting, we need to discuss two features that we haven't mentioned yet: how to perform backpropagation if the result depends on the dependent variable along different computation paths, and how to take derivatives when the variables aren't scalars.

44

modules / layers



45

Most deep learning frameworks also have a way of combining model parameters and computation into a single unit, often called a **module** or a **layer**.

In this case a Linear module (as it is called in Pytorch) takes care of implementing the computation of a single layer of a neural network (sans activation) and of remembering the weights and the bias.

break



source: <http://colah.github.io/>

making it work

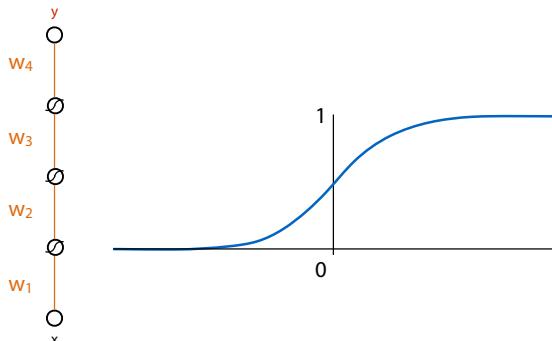
How to make *deep* neural nets work.

- Overcoming vanishing gradients
Proper initialisation, ReLUs over sigmoids
- Minibatch gradient descent
- Optimizers
(Nesterov) momentum, Adam
- Regularizers
L1, L2, Dropout

These are the four most important tricks that we use to train neural networks that are big (many parameters) and deep (many layers). Consequently, they are also the main features of any deep learning system.

47

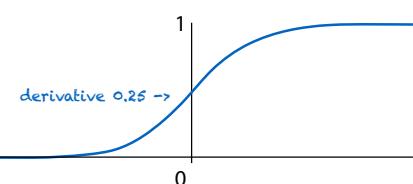
vanishing gradient problem



48

Here is a simple network to illustrate the problem of **vanishing gradients**. The question is how should we initialize its weights? If we set them too large, the activations will hit the rightmost part of the gradient. Consequently, the local gradient for each node will be very close to zero. That means that the network will never start learning.

If we go the other way, and make the weights large negative numbers, then we hit the leftmost part of the sigmoid and we have the same problem.

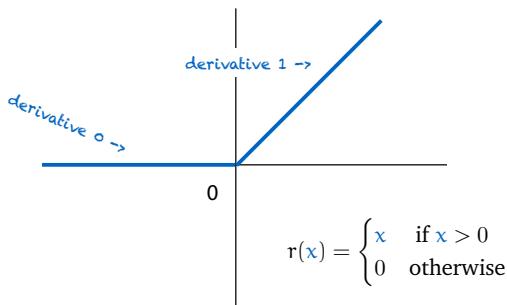


49

Even if the value going in to the sigmoid is close enough to zero, we still end up with a derivative of only one quarter. This means that propagating the gradient down the network, it will still go to zero with many layers.

We could fix this by squeezing the sigmoid, so its derivative is 1, but it turns out there is a better and faster solution that doesn't have any of these problems.

ReLU



50

The ReLU activation preserves the derivatives for the nodes whose activations it lets through. It kills it for the nodes that produce a negative value, of course, but so long as your network is properly initialised, about half of the values in your batch will always produce a positive input for the ReLU.

There is still the risk that during training, your network will move to a configuration where a neuron always produces negative input for every instance in your data. If that happens, you end up with a dead neuron: its gradient will always be zero and no weights below that neuron will change anymore (unless they also feed into a non-dead neuron).

initialization

Make sure your data is standardized/normalized
i.e. the mean is close to 0, and the variance is close to 1 in every direction.

initialise weights \mathbf{W} :

- Make \mathbf{W} a random orthogonal matrix (eigenvalues all 1).
- Glorot uniform:

$$w_{ij} \sim U\left(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \sqrt{\frac{6}{n_{in} + n_{out}}}\right)$$

There are two standard initialisation mechanisms. The idea of both is that we assume that the layer input is (roughly) distributed so that its mean is 0 and the variance is 1 in every direction (we must standardise or normalise the data so this is true for the first layer).

The initialisation is then designed to pick a random matrix that keeps these properties true (in a stochastic sense).

51

making it work

How to make *deep* neural nets work.

- ~~Overcoming vanishing gradients~~

Proper initialisation, ReLUs over sigmoids

- Minibatch gradient descent
- Optimizers
(Nesterov) momentum, Adam
- Regularizers
L1, L2, Dropout

These are the four most important tricks that we use to train neural networks that are big (many parameters) and deep (many layers). Consequently, they are also the main features of any deep learning system.

52

minibatch gradient descent

Like stochastic gradient descent, but with small batches of instances, instead of single instances.

Smaller batches: more like stochastic GD, **more noisy, less parallelism**.

Bigger batches: more like regular GD, **more parallelism**, limit is (GPU) memory

General advice, keep it between 16 and 128 instances.

53

making it work

How to make *deep* neural nets work.

- ~~Overcoming vanishing gradients~~

Proper initialisation, ReLUs over sigmoids

- ~~Minibatch gradient descent~~

- Optimizers
(Nesterov) momentum, Adam
- Regularizers
L1, L2, Dropout

These are the four most important tricks that we use to train neural networks that are big (many parameters) and deep (many layers). Consequently, they are also the main features of any deep learning system.

54

optimizers

Attempt to adapt the gradient descent update rule to improve convergence.

- Momentum
- Nesterov momentum
- RMSProp
- AdaGrad
- AdaDelta
- AdaMax
- Adam
- Nadam

General advice, use Adam, try Nesterov momentum if it doesn't work.

55

momentum

plain gradient descent

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \text{loss}(\mathbf{w})$$

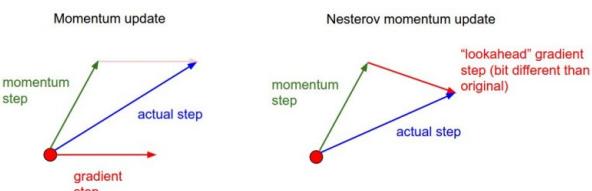
with momentum

$$\begin{aligned}\mathbf{v} &\leftarrow \mu \mathbf{v} - \eta \nabla \text{loss}(\mathbf{w}) \\ \mathbf{w} &\leftarrow \mathbf{w} + \mathbf{v}\end{aligned}$$

56

If gradient descent is a hiker in a snowstorm, then momentum gradient descent is a boulder rolling down a hill. The gradient doesn't affect its movement directly, it acts as a **force** on a moving object. If the gradient is zero, the updates continue in the same direction as the previous iteration, only slowed down by a "friction constant" mu.

Nesterov momentum



source: <http://cs231n.github.io/neural-networks-3/#sgd>
(recommended reading)

57

Nesterov momentum is a slight tweak. In regular momentum, the actual step taken is the sum of two vectors, the momentum step (a step in the direction we took last iteration) and a gradient step (a step in the direction of steepest descent at the current point).

Since we know that we are taking the momentum step anyway, we might as well evaluate the gradient *after* the momentum step. This will make the gradient slightly more accurate.

adam

Normalize the gradients: keep a running mean \mathbf{m} and (uncentered) variance \mathbf{v} for each parameter over the gradient. Subtract these instead of the gradient.

$$\begin{aligned}\mathbf{m} &\leftarrow \beta_1 * \mathbf{m} + (1 - \beta_1) \nabla \text{loss}(\mathbf{w}) \\ \mathbf{v} &\leftarrow \beta_2 * \mathbf{v} + (1 - \beta_2) (\nabla \text{loss}(\mathbf{w}))^2 \\ \mathbf{w} &\leftarrow \mathbf{w} - \eta \frac{\mathbf{m}}{\sqrt{\mathbf{v}} + \epsilon}\end{aligned}$$

Comes with three extra hyperparameters, but the defaults are usually fine.

58

Adam combines the idea of momentum with the idea that in large, complex networks each weight should have its own learning rate. Different weights perform very different functions, so ideally we want to look at the properties of the loss landscape for each weight (the sizes of recent gradients) and scale the "global learning rate" by these.

The bigger the recent gradients, the bigger we want the learning rate to be. However, if there is a lot of variance in the recent gradients, we want to reduce the learning rate (because the landscape is unpredictable). Thus, if we scale the learning rate by the mean \mathbf{m} over the recent gradients, and divide that by the square root of the variance \mathbf{v} (plus some small epsilon to avoid division by zero), we end up with a direction that uses recent information about the loss landscape to adapt the gradient to the local topology.

m and **v** are computed as an exponential moving average. This means that the current gradient weights the most, and the influence of recent gradients decays exponentially (but all play *some* part in the total sum).

making it work

How to make *deep* neural nets work.

- **Overcoming vanishing gradients**

Proper initialisation, ReLUs over sigmoids

- **Minibatch gradient descent**

- **Optimizers**

(Nesterov) momentum, Adam

- **Regularizers**

L1, L2, Dropout

These are the four most important tricks that we use to train neural networks that are big (many parameters) and deep (many layers). Consequently, they are also the main features of any deep learning system.

59

regularisers

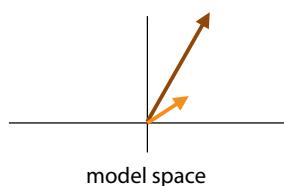
The bigger your model, the greater the capacity for **overfitting**.

Regularizers attempt to pull the model back towards more simple models, without eliminating the more complex solutions.

60

L2 regularizer

$$\text{loss}_{\text{reg}} = \text{loss} + \lambda \|\theta\|$$



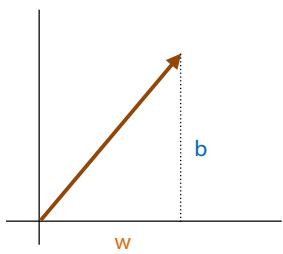
61

The L2 regularizer considers models with small parameters to be simpler (and therefore preferable). It adds a penalty to the loss for models with larger weights.

To implement the regularizer we simply compute the L2 norm of all the weights of the model (flattened into a vector). We then add this to the loss multiplied by hyper parameter **lambda**. Thus, models with bigger weights get a higher loss, but if it's worth it (the original loss goes down enough), they can still beat the simpler models.

Theta is a vector containing all parameters of the model (it's also possible to regularise only a subset).

vector norm

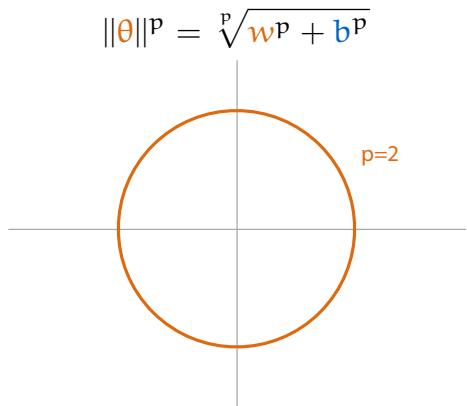


$$\theta = \begin{pmatrix} w \\ b \end{pmatrix}$$

$$\|\theta\| = \sqrt{w^2 + b^2}$$
$$\|\theta\|^p = \sqrt[p]{w^p + b^p}$$

We can generalise the l2 norm to an lp norm by replacing the squares with some other number p.

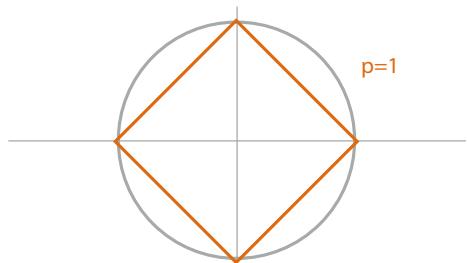
lp norm



For the l2 norm, the set of all points that have distance 1 to the origin form a circle.

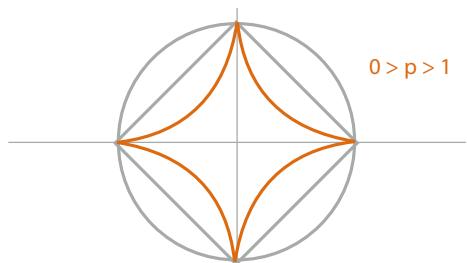
lp norm

$$\|\theta\|^p = \sqrt[p]{w^p + b^p}$$



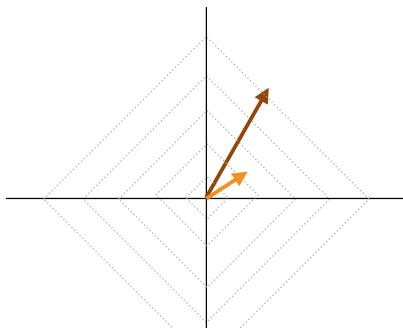
For the l1 norm, the form a diamond.

lp norm



L1 regularizer

$$\text{loss} \leftarrow \text{loss} + \lambda \|\theta\|^1$$



This means that for low p values ($p=1$ is most common), we get more sparse weight matrices: that is, if a weight is close to 0, it's more likely to get set entirely to zero.

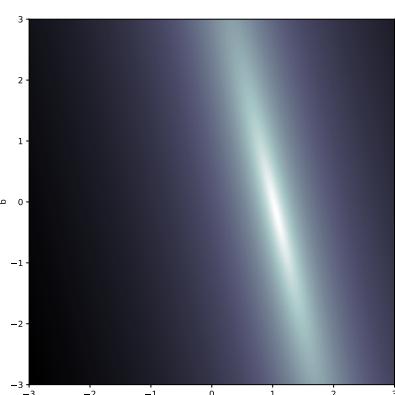


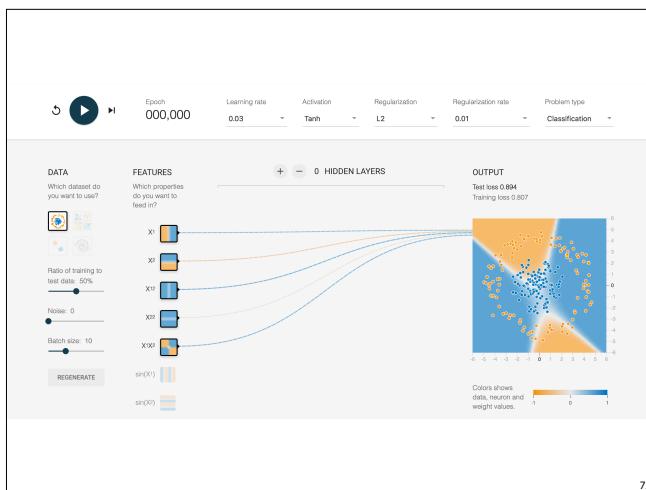
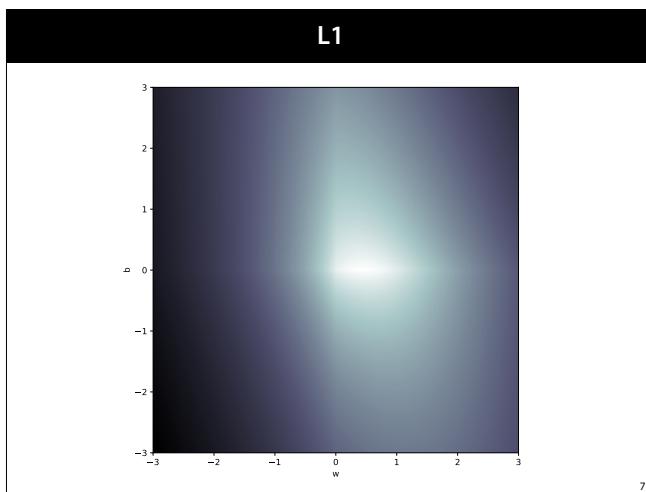
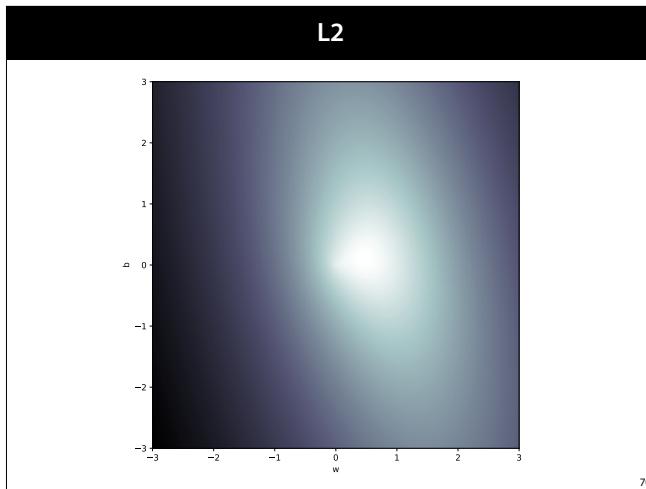
Here's an analogy. Imagine you have a bowl, and you roll a marble down it to find the lowest point. Applying L2 loss is like tipping the bowl slightly to the right. You shift the lowest point in some direction (like to the origin).



L1 loss is like using a square bowl. It has grooves along the dimensions, so that when you tip the bowl, the marble is likely to end up in one of the grooves.

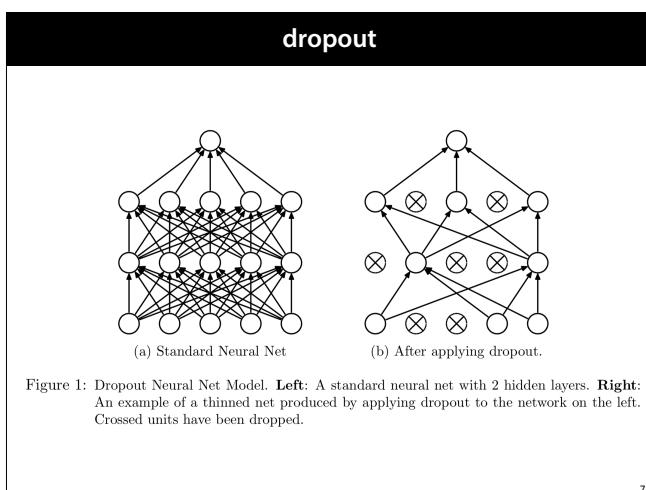
unregularized





We can try this in tensor flow playground. For this example (a simple logistic regression) we know that the derived features x_1^2 and x_2^2 contain everything we need to a linear fit. However, when we with with regularly, or with L2 regularization, we see that the weights for the other features never quite go to zero. However, with L1 regularization, we see that they become precisely zero.

<https://bit.ly/32lc6cQ>



Dropout is another regularization technique for large neural nets. During training, we simply remove hidden and input nodes (each with probability p).

This prevents co-adaptation. Memorization (aka overfitting) often depends on multiple neurons firing together in specific combinations. Dropout prevents this.

image source: <http://jmlr.org/papers/v15/srivastava14a.html>

dropout

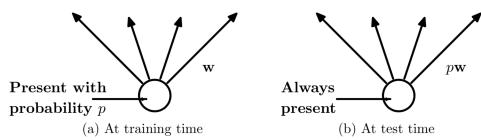


Figure 2: **Left:** A unit at training time that is present with probability p and is connected to units in the next layer with weights w . **Right:** At test time, the unit is always present and the weights are multiplied by p . The output at test time is same as the expected output at training time.

Once you've finished training and you start using the model, you turn off dropout. Since this increases the size of the activations, you should correct by a factor of p .

Frameworks like Keras know when you're using the model to train and when you're using it to predict, and turn dropout on and off automatically.

74



Following

If you ever need a definition of dropout that is both concise and accurate:

Dropout (Srivastava et al., 2014) may be the first instance of a human curated artisanal regularization technique that entered wide scale use in machine learning. Dropout, simply described, is the concept that if you can learn how to do a task repeatedly whilst drunk, you should be able to do the task even better when sober. This insight has resulted in numerous state of the art results and a nascent field dedicated to preventing dropout from being used on neural networks.

11:11 PM - 31 Mar 2018

215 Retweets 653 Likes



0 10 215 653

<https://twitter.com/Smerity/status/980175898119778304>

980175898119778304

75

summary: regularization

Provides a soft preference for "simpler" models.

L2: Simpler means smaller parameters

L1: Simpler means smaller parameters and more zero parameters

Dropout: randomly disable hidden units. Simpler means more robust

Many other tricks available for regularisation.

76

making it work

How to make *deep* neural nets work.

~~Overcoming vanishing gradients~~

Proper initialisation, ReLUs over sigmoids

~~Minibatch gradient descent~~

~~Optimizers~~

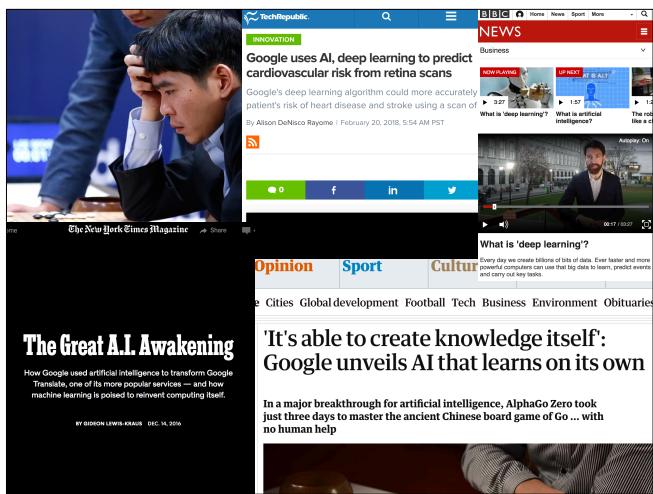
(Nesterov) momentum, Adam

~~Regularizers~~

L_1, L_2 , Dropout

These are the four most important tricks that we use to train neural networks that are big (many parameters) and deep (many layers). Consequently, they are also the main features of any deep learning system.

77



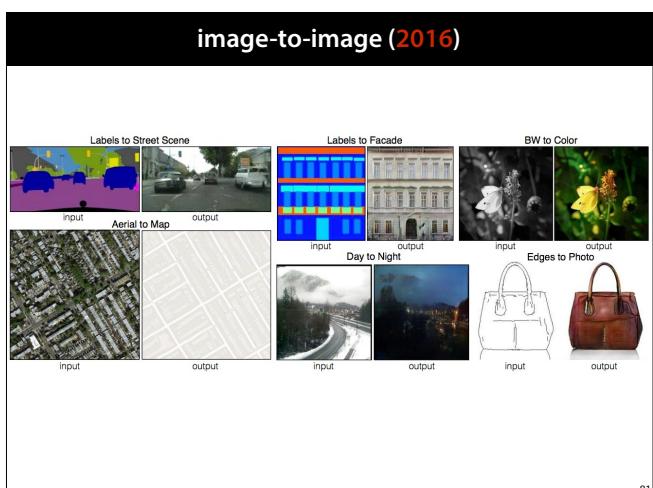
Once we had the basic frameworks for deep learning worked out and we started to get the hang of training big and deep networks, the deep learning revolution started to get going. Let's look at some early successes (mostly in the visual domain).



This is an end-to-end system for producing natural language descriptions of photographs. The system is not provided with any knowledge of the way language works, it just learns to produce captions from examples using a single neural network that consumes images and produces text, trained end-to-end.



This example uses a convolutional network to transfer the style of one image onto another. Interestingly, this work was done with a general purpose network, trained on a general classification task (such networks are available for download). The authors took this network, and didn't change the weights. They just built the style transfer architecture around the existing network.



This is pix2pix: a network with images as inputs and images as outputs was trained on various example datasets. Note the direction of the transformation. For instance, in the top left, the street scene with labeled objects was the *input*. The car-like objects, road surface, tree etc were all generated by the neural network to fill in the coloured patches in the input. Similarly, the bottom right shows the network generating a picture of a handbag from a *line-drawing*.

unmatched image to image (2017)



82

In some cases, we don't have neatly paired images: like the task of transforming a horse into a zebra. We can get a big bag of pictures of horses, and a big bag of pictures of zebras, but we don't know what a specific horse should look like as a zebra. The CycleGAN, published in 2017, could learn in this setting.

<http://gntech.ae/news/cyclegan-ai-can-turn-classic-paintings-photos/>



Ian Goodfellow
@goodfellow_ian

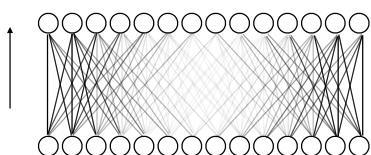
Following

4.5 years of GAN progress on face generation. arxiv.org/abs/1406.2661 arxiv.org/abs/1511.06434 arxiv.org/abs/1606.07536 arxiv.org/abs/1710.10196 arxiv.org/abs/1812.04948



https://twitter.com/goodfellow_ian/status/1084973596236144640

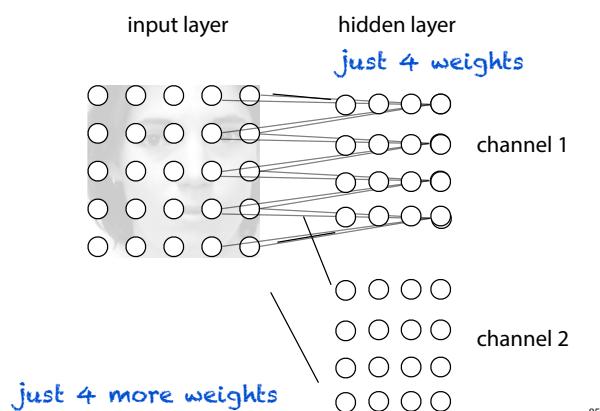
convolutional layers: pruning and weight sharing



To get started with deep learning, let's look at our first special layer. That is, a layer that is now just a fully connected linear transformation of a vector, but a layer whose shape is determined by some knowledge about its purpose. In the case of the convolution, its purpose is to consume *images*.

We know that images form a grid, and we can use this information to get far fewer connections, and far fewer weights in the layer than a fully connected layer.

Convolutional neural network

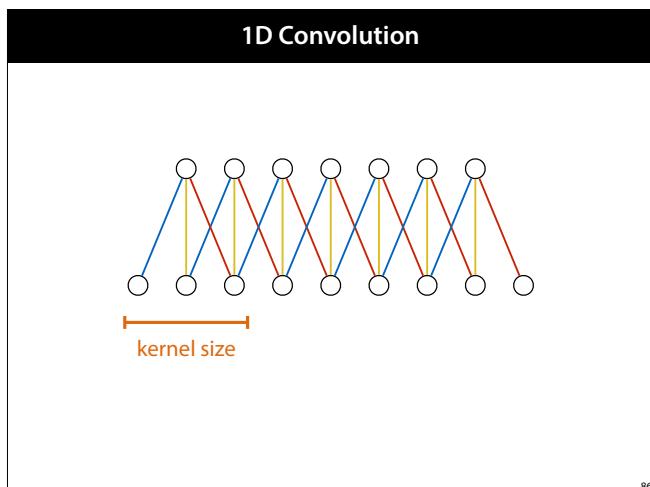


85

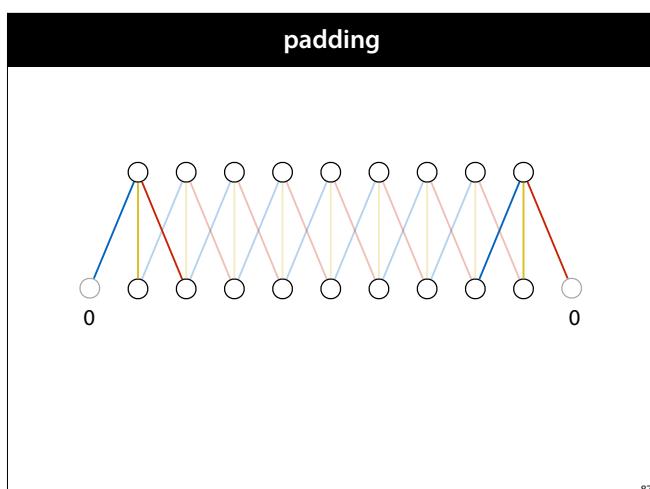
We connect each node in the hidden layer just to a small n by n neighbourhood in the input (here $n=2$); there are no connections to any other pixels. We do this for each such $n \times n$ neighbourhood in the input. For an input image of 5 by 5 pixels, this gives us an input layer of 25 nodes, and a hidden layer of 16 nodes (which we've also arranged in a grid). Each node in the hidden layer has just 4 incoming connections. What's more, we set the 4 weights of these incoming connections to be the same for each of the 16 nodes in the hidden layer.

To extend the hidden layer, we can add additional **channels** to the hidden layer. For an extra channel we follow the same procedure but with 4 new weights. If, as shown here, we have a 5 by 5 input layer with 4 pixel neighbourhoods, and two maps, we get a network with 25 inputs and 32 nodes in the hidden layer. In a traditional feedforward network, that would give us

25 x 32 connections with as many weights. Here, we have just 32 x 4 connections, and only 8 weights.



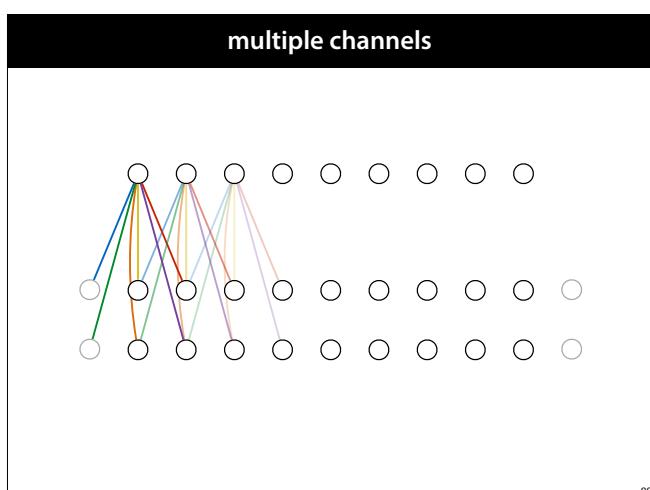
Here is how it looks if the input is 1D (a sequence of units rather than a grid). Note that the connection colors indicate shared weights (that is, every blue connection has the same weight).



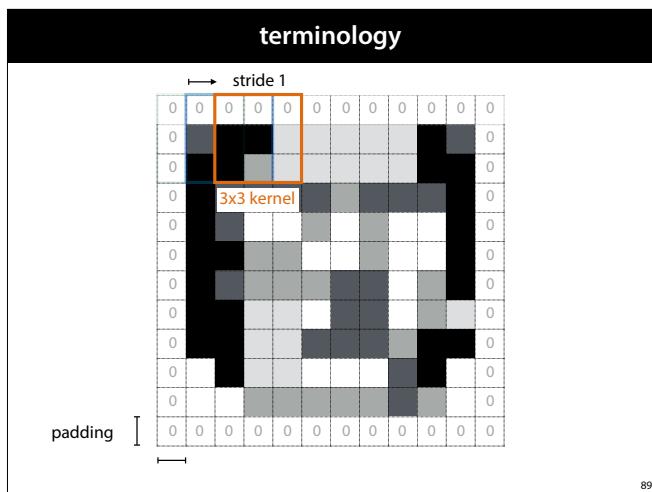
On drawback with the previous picture is that the inputs on the side contribute to only one hidden unit, and the one next to it to only two

To combat this, we can add *padding*: extra units, usually with a fixed value set to zero. Because of this padding, the number of output is the same as the number of inputs, and the actual units on the side contribute to more nodes.

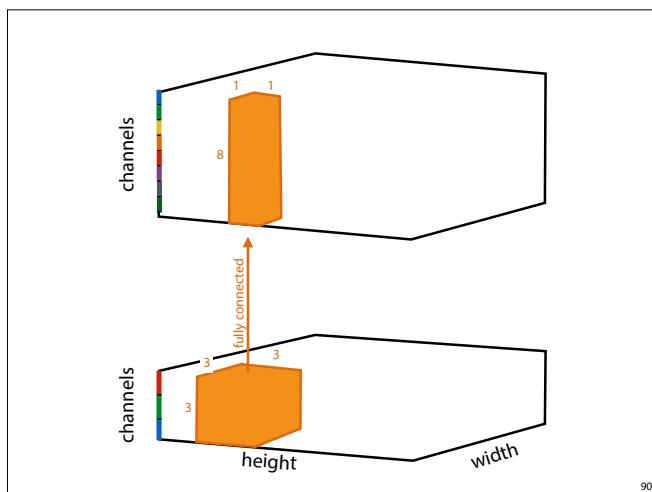
To achieve the same number of units in the output as in the input (before padding), we must set the number of units padded on both sides to $\text{floor}(\text{kernel_size}/2)$. This is sometimes called “same padding”.



If our input has multiple channels (like one color channel for each pixel), the standard approach is to add new weights for the new channel. Note that these are repeated along the spatial dimension(s) just like the other weights. The same approach is used to create multiple output channels.

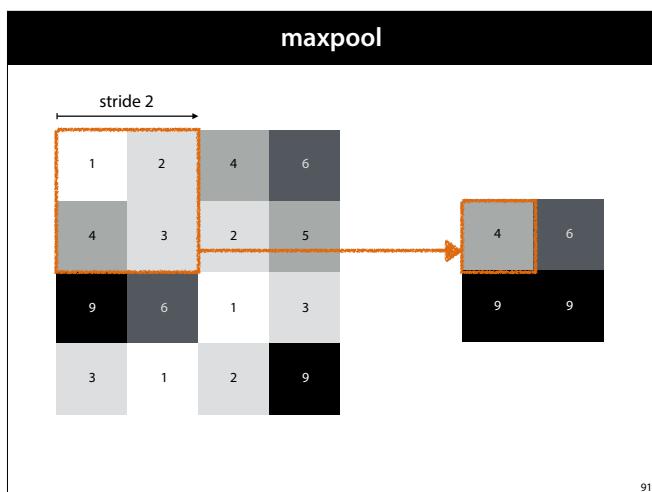


Here is the view in 2 dimensions. We normally slide the kernel one step each time (this is called a stride of 1), but we can also increase the stride to lower the output resolution.



Used in this way, the convolution layer transforms the input, a 3-tensor, into another 3-tensor with the same resolution and more channels.

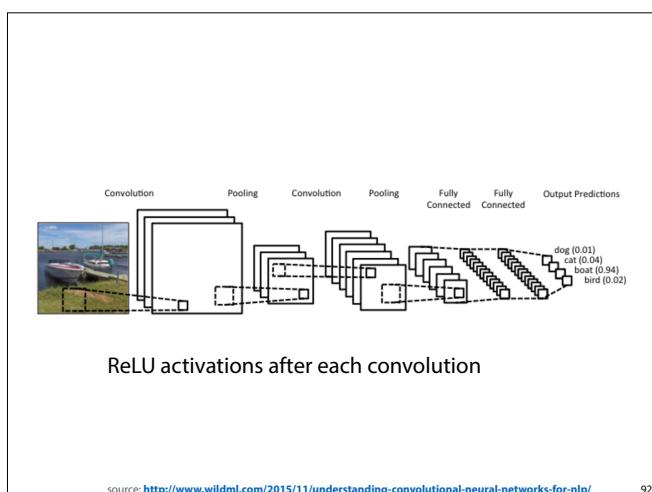
Between the two orange boxes, everything is fully connected (every channel of every pixel in the lower boxer is connected by unique weight to every channel of every pixel in the top box).



We chain these convolutions together, but after a while (as the number of channels grows) we'd like the resolution to decrease so we're gradually looking at less specific parts of the image, but we have more information (more channels) about that part of the image.

The max pooling layer does this for us, it divides the image into n-by-n squares, and returns the maximum value from each square. Average pooling (returning the average over each square) is also possible, but max pooling is usually more effective.

Most likely, this is because during the backward pass, the whole gradient flows back down one of the pixels, instead of dividing over all four. This gives the backpropagation a sparse character: it focuses only on the most important paths in the computation graph instead of dividing its attention over all of them.



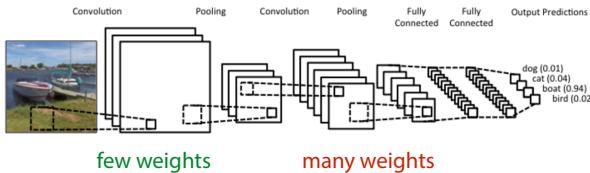
We extend this principle for many more layers. At each step the maps of the layers get smaller, and we add more maps.

Eventually, we add one or two fully connected layers, and a (softmax) output layer (if we're doing image classification).

more channels = more weights

high resolution, low channels: **few weights**

low resolution, many channels: **many weights**



93

worksheet 4

```
from keras.models import Sequential
from keras.layers import Dense, Activation, Conv2D, MaxPool2D, Dropout, Flatten

model = Sequential()
model.add(Conv2D(16, kernel_size=(3, 3), activation='relu', input_shape=(28,28,1)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25)) # Dropout 25% of the nodes of the previous layer during training
model.add(Flatten()) # Flatten, and add a fully connected layer
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax')) # Last layer: 10 class nodes, with dropout
model.summary()

from keras.optimizers import Adam

optimizer = Adam()
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model, iterating on the data in batches of 32 samples
model.fit(x_train, y_train, epochs=15, batch_size=32, validation_split=1/6)
```

94

filters



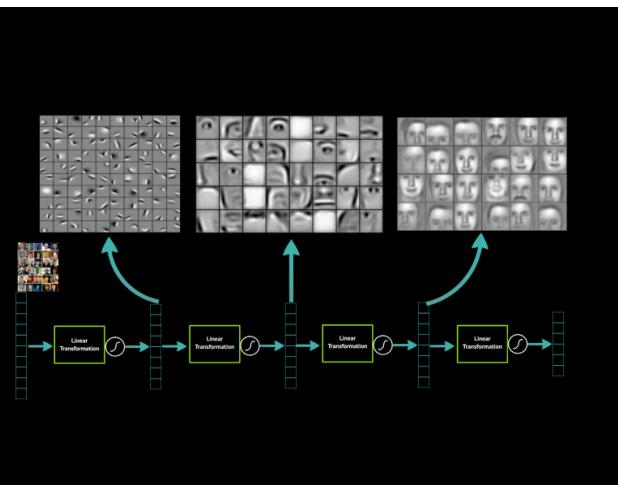
95

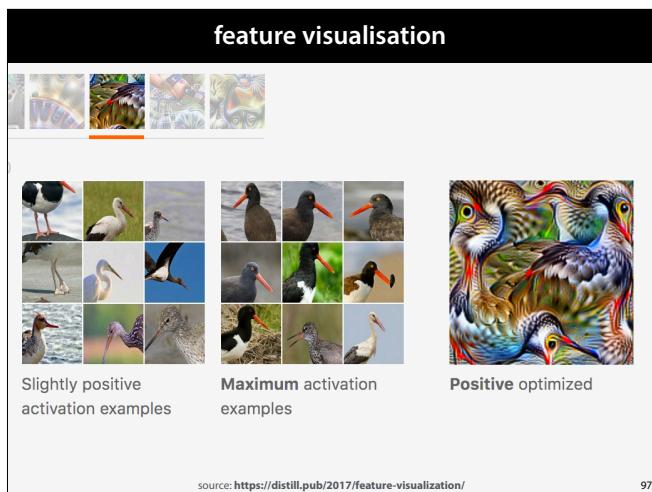
Each first layer gives us several **convolution filters** (one per channel). Here is an example: the Gaussian convolution. It takes a pixel neighbourhood and averages the pixels in it, creating a blurred result of the input (this is just one transformation that a convolution filter can perform, depending on the weights, many other operations are possible).

While Gaussian blur may seem to be throwing away valuable information, what we actually get is a representation that is *invariant* to noise. All these noisy input images in the left will be mapped to the same image on the right. We can do the same thing to create representations invariant to, for instance, small translation.

Here are the results of a real convolutional network trained to detect faces. The small grayscale images show a typical image that each node in one of the layers responds to. Those for the first layers can be thought of as edge detectors: if there is a strong edge in a particular part of the image, the node lights up. The second combine these into detectors for parts of images: eyes noses, mouths, etc. The third combine these into detectors for complete faces.

source: <https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-core-concepts/>

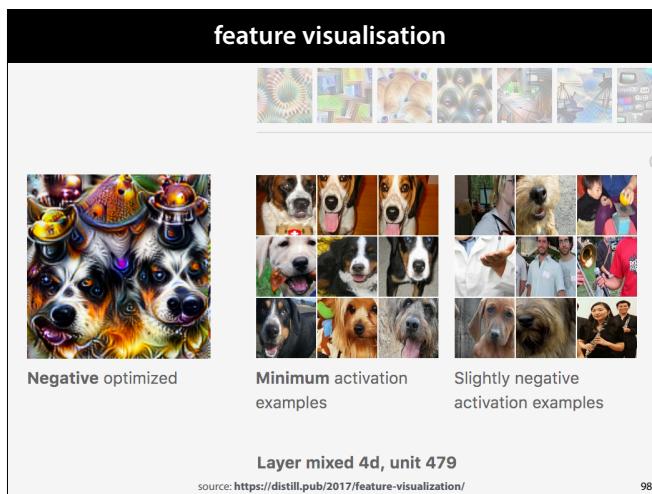




Here is a feature visualisation example for a more recent network trained on Imagenet, a collection of 14 million images with diverse subjects.

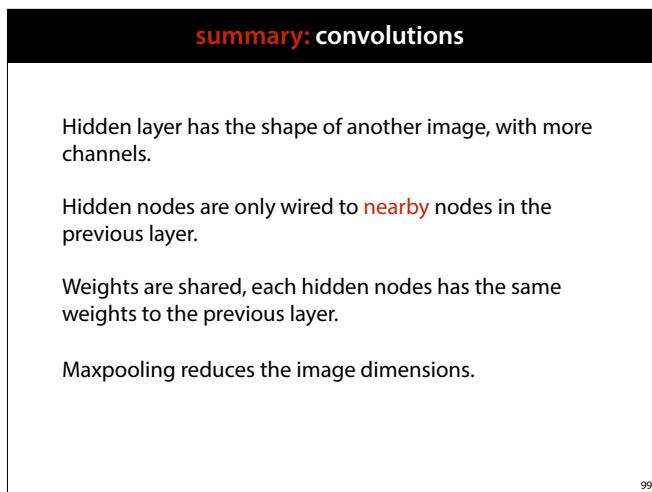
To find the image on the right, the authors took one node high up in the network, and instead of optimising the weights to minimise the loss, they optimised the input to maximise the activation of that node.

They also searched the dataset for natural images that caused a high activation in that particular node.



The opposite is also possible: searching for an input that cause minimal activation.

<https://distill.pub/2017/feature-visualization/>

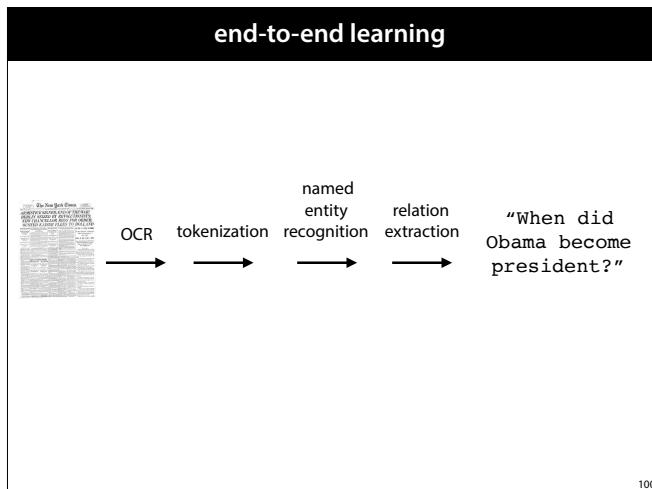


Finally, let's discuss what deep learning means on a higher level; why we consider it such a departure from classical machine learning.

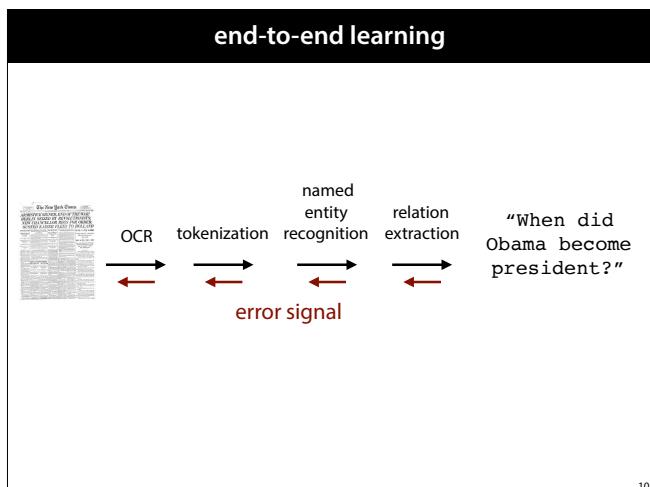
Here is the kind of pipeline we would often attempt to build in the days before deep learning: we scan old news papers, perform optical character recognition, tokenise the characters into words, attempt to find named entities (like people and companies) and then try to learn the relations between these entities so that we can ask structured queries.

Most of these steps would be solved by some form of machine learning. And after a while, we were getting pretty good at each. So good that it would, for instance, make a mistake for only 1 in a 100 instances.

But chaining together modules that are 99% accurate does not

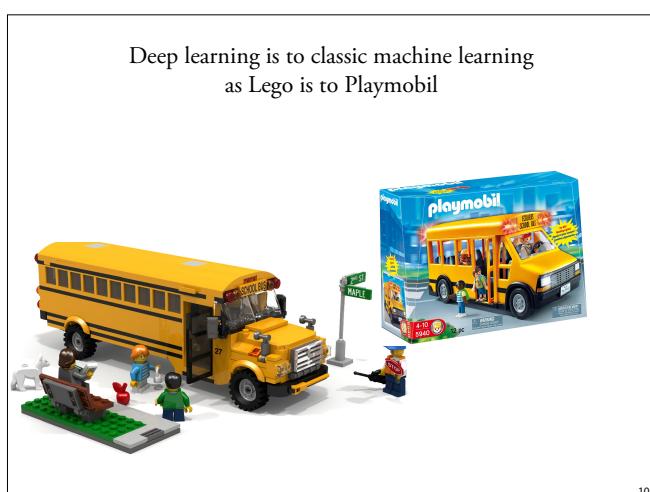
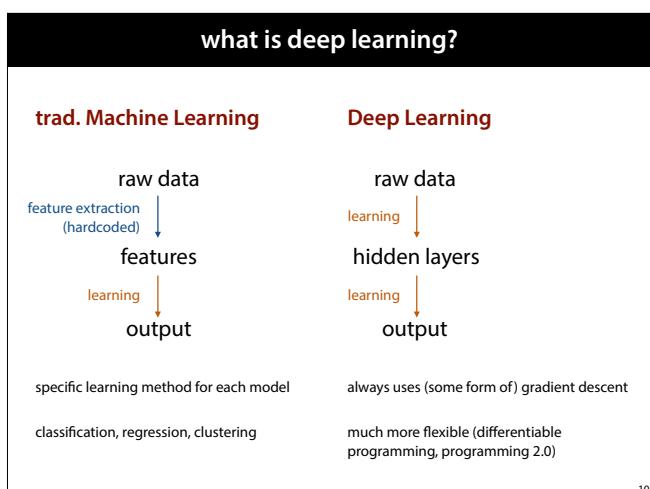


give you a pipeline that is 99% accurate. Error *accumulates*. The tokenization works slightly less well than on its pristine test data, because it's getting noisy input from the OCR. This makes the input for the NER module even more noisy and so on.



The solution of **deep learning** is to make each module differentiable: ensure that we can work out a local gradient so that we can also train the pipeline as a whole using backpropagation.

This is called **end-to-end learning**. Other names for roughly the same idea are **differentiable programming** and **programming 2.0**.



It's a lower level of abstraction, which gives you smaller building blocks. This allows us more creativity in what we can build.

mlcourse@peterbloem.nl
